数据字典

Activity.h/.cpp

- `class activity`：活动类
    - `bool isPrivate`：是否是公开活动
    - `int week`：活动所在周数
    - `int locationID`：地点编号
    - `String activityName`：活动名
    - `String locationName`：地点名

Array.cpp

- `class Array`：一维数组类
    - `T* data`：数据存放地址指针
    - `int maxN`：数组长度

AVL.cpp

- `class AVLNode`：AVL树节点类
    - `T data`：节点上的数据
    - `int size`：子树大小
    - `int deep`：子树最大深度
    - `AVLNode* left`：左孩子指针
    - `AVLNode* right`：右孩子指针
- `class AVL`：AVL树类
    - `AVLNode* root`：树根节点指针

Basic.cpp

- `struct Less`：比较类，重载了()运算符，lhs < rhs 时返回 true
- `struct Greater`：比较类，重载了()运算符，lhs > rhs 时返回 true
- `struct LessEqual`：比较类，重载了()运算符，lhs <= rhs 时返回 true
- `struct GreaterEqual`：比较类，重载了()运算符，lhs >= rhs 时返回 true
- `void swapElement(T *x, T *y)`：交换元素 x 和 y
- `void sort(T *a, int len);`：快速排序
- `T getMax(T x, T y)`：获取较大值
- `struct Time`：时间类

Calendar.h/.cpp

- `class Calendar`：日程表类
    - `int size`：当前日程表

Campus.h/.cpp

- `struct Campus`：校区类
    - `Graph G`：校区对应的图

Course.h/.cpp

- `class Test`：考试类
    - `Pair<Time, Time> testTime;`：考试起止时间
    - `int cid`：考试对应课程编号
- `class Course`：课程类
    - `int courseid`：课程编号
    - `String courseName`：课程名
    - `int location`：上课地点所在建筑编号
    - `int campus`：上课地点所在校区
    - `String classroom`：上课教室
    - `String teacher`：授课老师
    - `String qq`：联系方式
    - `Pair<Time, Time> courseTime`：课程起止时间
    - `Pair<int, int> courseWeek`：课程起止周数
    - `Test test`：考试信息
    - `Vector<String> handIn`：已交作业
    - `Vector<String> toBeHandIn`：待交作业
    - `Vector<itinerary*> events`：课程的所有活动事项
- `struct Course_ptr`：课程指针类，封装并重载了大小比较，方便 AVL 树使用
- `class CourseSys`：课程管理类
    - `AVL<Course_ptr> courseTree`：维护所有课程的AVL树，以课程名为关键字，方便按名称查询
    - `int cnt`：课程数
    - `Vector<Course*> courseArray`：所有课程列表

DyadicArray.cpp

- `class DyadicArray`：二维数组类
    - `T* data`：数据起始地址指针
    - `int maxN`：最大行数
    - `int maxM`：最大列数

EdgeNode.h/.cpp

- `struct EdgeNode`：边类
  - `double dis`：边的长度
  - `double crowdDegree`：拥挤程度
  - `bool isBike`：是否允许自行车通行
- `struct WalkCalc`：边权计算类，重载()运算符，用于计算步行通过某条边时间
- `struct BikeCalc`：边权计算类，重载()运算符，用于计算骑车通过某条边时间
- `struct DisCalc`：边权计算类，重载()运算符，用于计算某条边距离

FileCompresser.h/.cpp

- `class TreeNode`：哈夫曼树节点类
  - `unsigned char c`：节点代表字符
  - `int weight`: 节点权重
  - `TreeNode* left`：左孩子指针
  - `TreeNode* Right`：右孩子指针
- `struct NodeCompare`：节点比较类，重载()运算符，用于比较节点
- `class HuffmanTree`：哈夫曼树类
  - `int64_t totalBits`：压缩后字符数
  - `int frequency[256]`：每个字符出现的频率数组
  - `TreeNode* root`：根节点
  - `Heap<TreeNode*, NodeCompare> priQueue`：堆实现的优先队列，用于建树
  - `std::string charCode[256]`：存储字符编码后结果

Graph.h/.cpp

- `class Graph`：地图类，采用链式前向星存储
  - `Array<int> head`：每个点对应的第一条边的编号
  - `Vector<int> end`：每条边对应的终点
  - `Vector<int> next`：每条边在链表上的下一条边编号
  - `Vector<EdgeNode> weight`：每条边的边权
  - `int vertexNum`：点数
  - `int edgeNum`：边数
  - `Route routes[11];`：所有最短路径
  - `Vector<int> pre[103]`：所有点在最短路径上的前驱点
  - `Vector<int> preEdge[103]`：所有点在最短路径上的前驱边

Guider.h/.cpp

- `struct CrossEdge`：跨校区边类
  - `int x, y, u, v`：从 x 校区点 u 到 y 校区点 v
  - `double time`：耗时
- `struct CrossTimeNode`：跨校区发车表

    - `CrossEdge e`：跨校区边详细信息
    - `int m`：发车时刻表长度
    - `Vector<double> timeTable`：发车时刻表
- `class Guider`：导航系统类
    - `int campusNum`：校区数
    - `int crossEdgeNum`：跨校区边数
    - `Vector<CrossTimeNode> crossEdge`：跨校区边列表
    - `Campus campusMap[2];`：校区列表
    - `Route tmpx[11]`：起始校区内的最短路径
    - `int xNum`：起始校区内的最短路径数
    - `Route tmpy[11]`：终止校区内的最短路径
    - `int yNum`：终止校区内的最短路径数
    - `double busTime`：跨校区最短路径对应的发车时间
    - `CrossEdge busEdge`：跨校区最短路径对应的跨校区边

Heap.cpp

- `class Heap`：堆类
    - `Vector<T> data`：堆中所有数据
    - `int size`：堆中元素个数

KMP.h/.cpp

- `class KMP`：封装 KMP 算法的类
    - `char* s`：待匹配字符串
    - `char* t`：用于匹配字符串
    - `int lens`：字符串 s 的长度
    - `int lent`：字符串 t 的长度
    - `bool exist`：指示 s 是否含有 t
    - `Vector<int> next`：KMP 算法自匹配得到的 next 数组
    - `Vector<int> ans`：KMP 算法求得的匹配位置

Material.h/.cpp

- `class Material`：文件（资料/家庭作业）类
    - `bool homework`：是否为家庭作业
    - `String courseName`：课程名称
    - `String name`：名称
    - `ByteArray md5`：MD5 码
    - `Time updateTime`：更新时间
    - `String path`：文件路径
    - `Material* nextVersion`：下一个版本指针
- `MaterialPtr`：文件类指针，重载了比较运算符
- `class MaterialSys`：文件管理系统
    - `Vector<Material*> materials`：文件指针数组，用于存储所有文件

    - `RBTree<MaterialPtr> materialTree`：以 MD5 码为关键字的红黑树，用于判重
    - `RBTree<Spair<String, int>> nameTree`：以名称为关键字的红黑树，用于快速按名称查询
    - `RBTree<Spair<Time, IntPair>> timeTree`：以时间为关键字的红黑树，用于快速按时间查询
    - `Vector<Material*> mForSort`：文件指针数组，用于排序

Pair.cpp

- `class Pair`：有序二元组，第一个元素为第一比较关键字，第二个元素为第二比较关键字
    - `T1 first`：第一个元素
    - `T2 second`：第二个元素
- `Typedef Pair<int, int> IntPair`：整型对
- `class Spair`：有序二元组，第一个元素为比较关键字，第二个元素不参与比较
    - `T1 first`：第一个元素
    - `T2 second`：第二个元素

rbtree.cpp

- `enum RBTColor`：红黑树颜色类型
- `struct RBTNode`：红黑树节点
    - `T data`：节点数据值
    - `RBTColor color`：节点颜色
    - `RBTNode* parent`：父节点指针
    - `RBTNode* left`：左孩子指针
    - `RBTNode* right`：右孩子指针
- `struct RBTree`：红黑树
    - `RBTNode<T> *root`：根节点指针

SegmentTree.h/.cpp

- `class itinerary`：日程类
    - `Pair<Time, Time> t`：起止时间
    - `String name`：日程名称
    - `int campus` ：所在校区编号
    - `int location`：所在建筑地点编号
    - `string room`：所在地点的字符串描述
    - `int type` ：设置日程类型（个人、集体、其他 = 0）
- `class segmentTree`：线段树类
    - `struct SegNode`：线段树节点
        - `bool value`：是否被占用标记
        - `bool purity`：是否属于同一日程标记
        - `itinerary* point`：对应日程的指针
    - `int lazy[35105]`：懒惰标记数组

- `SegNode timeSegment[35105]`：线段树节点数组

String.h/.cpp

- `class StringNode`：字符串节点
  - `char c`：该节点字符值
  - `StringNode* next`：链表指示下一个元素的指针
- `int size`：字符串长度
- `StringNode* head`：字符串头指针
- `StringNode* tail`：字符串尾指针

Vector.cpp

- `class Vector`：变长数组类
  - `T* data`：数据存放地址指针
  - `int maxLength`：数组最大长度上限
  - `int size`：数组长度

数据结构

线性表

一维数组 Array

说明：通过泛型编程实现的模板类，采用指针实现的一维定长数组，设置了3中不同的构造函数满足不同场景下的不同需求，同时对下标运算符和赋值运算符进行了重载，考虑到深拷贝以及自赋值问题，保证该类运行时稳定不出错。
成员函数声明如下：

```
template<class T>
class Array {
 private:
    T *data;
    int maxN;
 public:
    Array();
    explicit Array(int n);
    Array(int n, T x);
    ~Array();
    T& operator[](const int &index) const;
    Array<T>& operator = (Array other);
};
```

其中，所有待存储的数据项存储在成员变量 T* data 中，数量为 maxN。其接口函数包括：

（1）构造函数

构造函数支持直接无参数构造数组、在已知数组长度的情况下构造数组、在已知数组长度和元素值的情况下构造数组。都是通过 malloc 函数申请内存空间实现的，具体实现参看代码部分：

```
template<class T>
Array<T> :: Array() {
    data = NULL;
}

template<class T>
Array<T> :: Array(int n) {
    data = reinterpret_cast<T*> (malloc(n * sizeof(T)));
    maxN = n;
}

template<class T>
Array<T> :: Array(int n, T x) {
    data = reinterpret_cast<T*> (malloc(n * sizeof(T)));
    maxN = n;
    for (int index = 0; index < n; index++)
        data[index] = x;
}
```

（2）重载运算符 []

重载运算符允许数组被按照下标直接访问，和 C 语言自身的数组类似。代码实现如下：

```
template<class T>
T& Array<T> :: operator[](const int &index) const {
    return data[index];
}
```

（3）重载运算符 =

重载运算符 = 可以比较两个数组是否相等，当且仅当它们长度相同且每个数组元素均相同时，才认为这两个数组相等，代码实现如下：

```
template<class T>
Array<T>& Array<T> :: operator = (const Array &other) {
    T *newData = reinterpret_cast<T*> (malloc(other.maxN * sizeof(T)));
    maxN = other.maxN;
    for (int index = 0; index < maxN; index++)
        newData[index] = other[index];
    data = newData;
    return *this;
}
```

二维数组 DyadicArray

说明：通过泛型编程实现的模板类，相较于一维数组增加了一个维度，是一维数组的扩展以满足更多的需求，其他方面与一维数组类似。

```
template<class T>
class DyadicArray {
 private:
    T *data;
    int maxN, maxM;
 public:
    DyadicArray();
    DyadicArray(int n, int m);
    DyadicArray(int n, int m, T x);
    ~DyadicArray();
    T* operator[](const int &index) const;
};
```

二维数组和一维数组没有太多差异，仅仅是申请空间时需要参数不同。略有差异的是重载运算符 []，它允许用户通过 DyadicArray[] [] 的形式访问数组元素，通过传入的下标参数和指针计算出对应数据项在内存中的位置，从而将对应数据项返回即可。代码实现如下：

```
template<class T>
T* DyadicArray<T> :: operator[](const int &index) const {
    return data + index * maxM;
}
```

测试用例及结果

对于一维数组与二维数组，我们以整型数据为例进行了测试，代码如下：

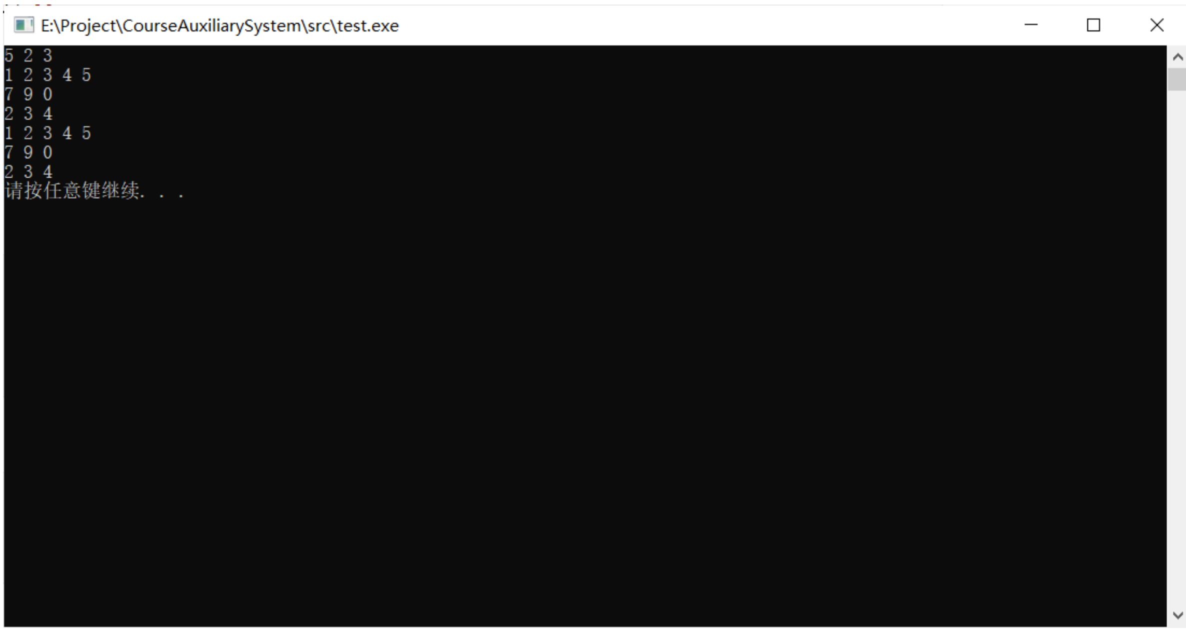
```
int main() {
    int n, m, k;
    cin >> n >> m >> k;
    Array<int> a(n, 0);
    for (int i = 0; i < n; i++)
        cin >> a[i];
    DyadicArray <int> c(m, k);
    for (int i = 0; i < m; i++)
        for (int j = 0; j < k; j++)
            cin >> c[i][j];
    Array<int> b = a;
    for (int i = 0; i < n; i++)
        cout << b[i] << " ";
    cout << endl;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < k; j++)
            cout << c[i][j] << " ";
        cout << endl;
    }
    system("pause");
    return 0;
}
```

输入：

```
5 2 3
1 2 3 4 5
7 9 0
2 3 4
```

输出：

```
1 2 3 4 5
7 9 0
2 3 4
```



串 String

说明

该类同样是对于C++ String类的仿写，为操控字符串提供便利，其本质上是一个限定数据类型为char的链表并有着头尾指针，在类中，我们实现了赋值以及小于运算运算符的重载，重载+运算符为连接运算符，同时编写了成员函数用于类型转换，查看串长度等，进一步为操控字符串提供了便利。

`int getSize(): 获取长度`

`void pushBack(char c): 将字符插入串尾`

`char* data():将String转换为char*`

```
class String {
 private:
    struct StringNode {
        char c;
        StringNode* next;
        StringNode();
        explicit StringNode(char x);
        ~StringNode();
    };
    int size;
    StringNode *head, *tail;
 public:
    String();
    explicit String(char* x);
    ~String();
    int getSize();
    void pushBack(char c);
```

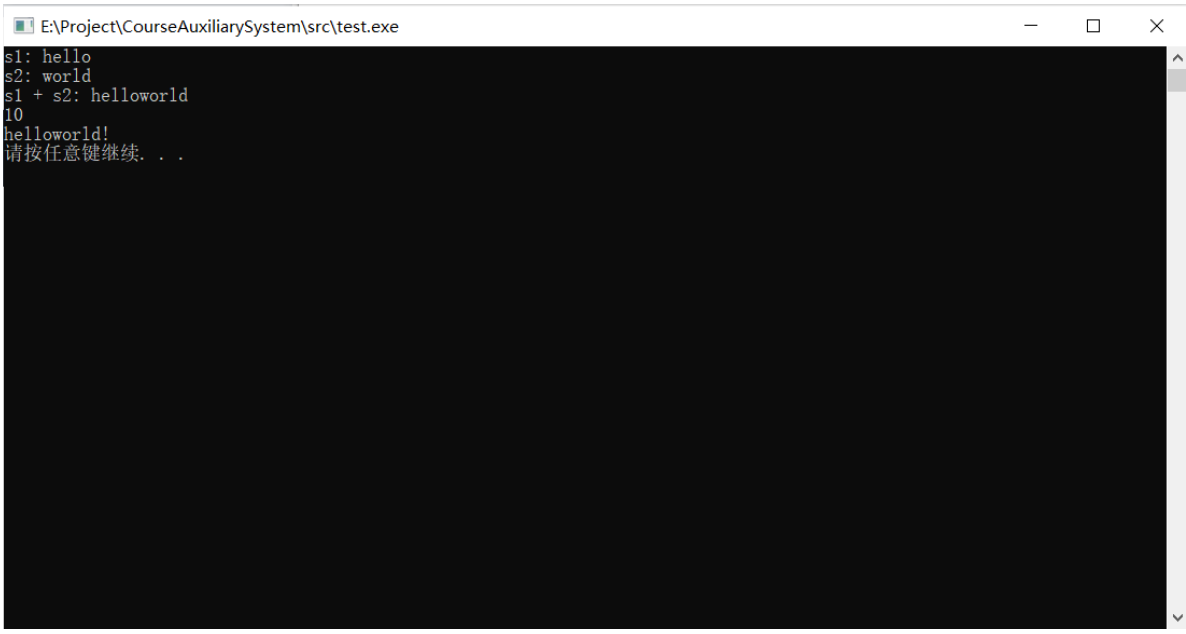
```
    char* data();
    String& operator = (String other);
    String operator + (String other);
    bool operator < (String other);
};
```

测试数据及结果

测试代码如下：

```
int main() {
    String s1, s2;
    s1 = "hello";
    s2 = "world";
    cout << "s1: " << s1.data() << endl;
    cout << "s2: " << s2.data() << endl;
    String s3 = s1 + s2;
    cout << "s1 + s2: " << s3.data() << endl;
    cout << s3.getSize() << endl;
    s3.pushBack('!');
    cout << s3.data() << endl;
    system("pause");
    return 0;
}
```

运行结果如下：



变长数组 Vector

说明

该类是对C++ STL中的Vector的仿写，是可以变长的数组，通过在设置不同的阈值，在超过某阶段阈值后重新开辟空间并将原有数据进行拷贝。同时提供数组反转，判断空数组等成员方法

```
template <class T>
```

```
class Vector {
 private:
    T *data;
    int maxLength;
    int size;
    //重新分配数组空间
    void reNew(int newLength);

 public:
    Vector();
    explicit Vector(int n);
    Vector(int n, T x);
    ~Vector();
    //获取数组元素数目
    int getSize();
    //添加新元素至数组尾部
    void pushBack(T e);
    //将数组末尾元素弹出
    void popBack();
    //判断数组是否为空
    bool isEmpty();
    T &operator[](int index);
    Vector<T>& operator = (Vector<T> other);
    Vector<T> operator + (Vector<T> other);
    //将数组逆序
    void reverse();
};
```

Vector 类主要在不能事先确定数组长度时，提供一种可变长数组的选项，通过核心的 reNew 函数重新分配空间，避免了数组长度过小导致数组越界，或数组长度过长导致占用过多内存资源的问题。其关键函数如下：

（1）更新长度 reNew

reNew 函数负责在原本分配的空间不足时，重新为数组动态申请内存空间。具体策略为：首先默认申请大小为 64 的数组空间；如果目前分配的空间全部用完，还有新的元素加入时，则重新申请一个长度为原本两倍的空间，将原本的数据项全部拷贝过去，释放原本的内存空间。具体代码实现如下：

```
template<class T>
void Vector<T>::reNew(int newLength) {
    T *newData = reinterpret_cast<T *>(malloc(newLength * sizeof(T)));
    memset(newData, 0, newLength * sizeof(T));
    for (int i = 0; i < size; i++) newData[i] = data[i];
    free(data);
    data = newData;
    maxLength = newLength;
}
```

（2）尾部插入函数 pushBack 和删除函数 popBack

pushBack 函数是尾部插入函数，首先会判断当前申请的内存空间是否已经被占满了，如果被占满了，则使用 reNew 函数更新当前的内存空间，再将元素插入数组的尾部；popBack 函数直接在尾部删除即可。代码实现如下：

```cpp
template<class T>
void Vector<T>::pushBack(T e) {
    if (size == maxLength) reNew(maxLength << 1);
    data[size++] = e;
}

template<class T>
void Vector<T>::popBack() {
    if (size == 0) return;
    size--;
}
```

（3）重载运算符 +

重载了+运算符，使得可以直接将一个 Vector 接在另一个 Vector 之后得到一个新的 Vector。具体实现是，将另一个 Vector 的每个元素依次 pushBack 到 Vector 中，代码实现如下：

```cpp
template<class T>
Vector<T> Vector<T>::operator+(Vector<T> other) {
    Vector<T> ans = *this;
    for (int index = 0; index < other.getSize(); index++)
        ans.pushBack(other[index]);
    return ans;
}
```

（4）其他接口函数

reverse 函数可以将 Vector 内的元素反转，具体而言，将 Vector 前后的元素依次使用 swapElement 函数交换即可。

getSize 函数可以返回 Vector 内的元素个数。

isEmpty 函数可以判断 Vector 是否为空，若为空，则返回1.

重载运算符 [] 可以用访问数组的方式访问 Vector 元素。

重载运算符 = 可以判断两个 Vector 是否相等，具体方法为，若 Vector 的每个元素都相等，则这两个 Vector 相等。

具体代码实现较为简单，可以直接参看源程序部分。

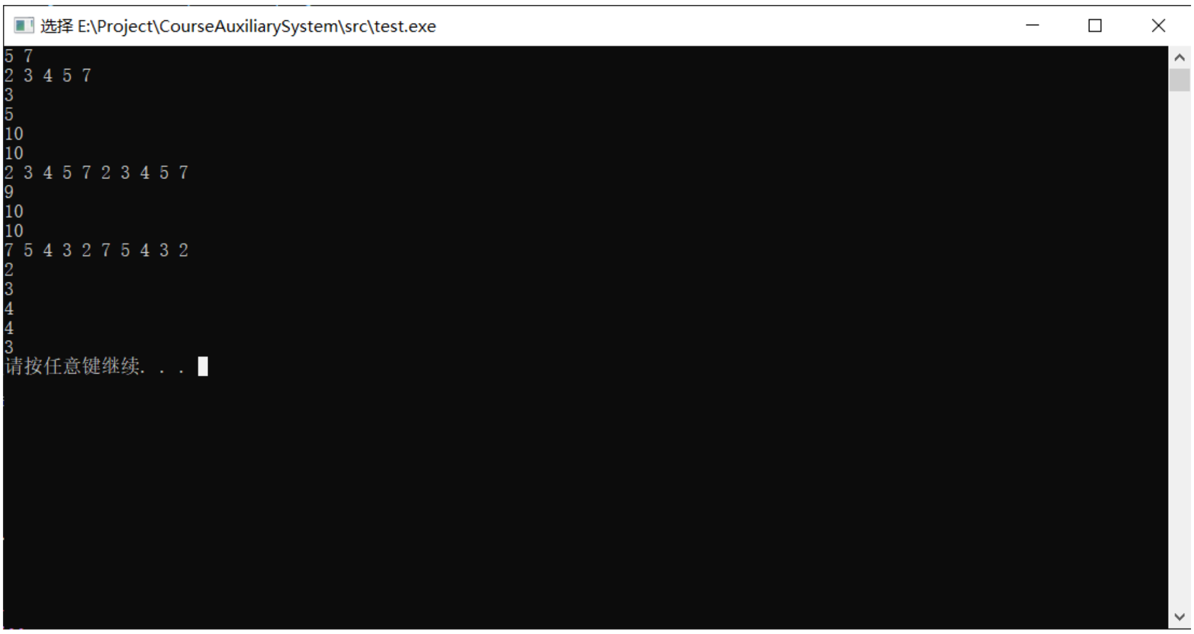
测试数据及结果

我们以整型数据为例进行了测试，代码如下：

```cpp
Vector<int> A;

int main() {
    int n, Q;
    cin >> n >> Q;
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        A.pushBack(x);
    }
    for (int i = 0; i < Q; i++) {
        int x;
        cin >> x;
```

```cpp
        if (x == 1) {
            int y;
            cin >> y;
            A.pushBack(y);
        } else if (x == 2) {
            A.popBack();
        } else if (x == 3) {
            cout << A.getSize() << endl;
        } else if (x == 4) {
            cout << A[A.getSize() - 1] << endl;
        } else if (x == 5) {
            A.clear();
        } else if (x == 6) {
            cout << A.isEmpty() << endl;
        } else if (x == 7) {
            int y;
            cin >> y;
            cout << A[y] << endl;
        } else if (x == 8) {
            int y;
            cin >> y;
            A[y] = A[A.getSize() - 1];
            A.popBack();
        } else if (x == 9) {
            A.reverse();
        } else if (x == 10) {
            Vector<int> B = A + A;
            cout << B.getSize() << endl;
            for (int i = 0; i < B.getSize(); i++) {
                cout << B[i] << " ";
            }
        }
    }
    return 0;
}
```

部分测试结果如下图：



线段树

算法描述

线段树是一种二叉搜索树，与区间树相似，它将一个区间划分成一些单元区间，每个单元区间对应线段树中的一个叶结点。使用线段树可以快速的查找某一个节点在若干条线段中出现的次数，时间复杂度为O(logN)。同时，为了进一步优化效率，只在需要查询的时候进行部分区间的修改而非即可修改整个区间，我们增加了lazy标记：对整个结点进行的操作，先在结点上做标记，而并非真正执行，直到根据查询操作的需要分成两部分。

类声明：

```
class segmentTree {
// 此线段树对时间区间的操作为左闭右闭，即活动开始时间也不能同结束时间相同
 private:
    int lazy[35105];
    struct SegNode {
        bool value;
        bool purity;
        itinerary* point;
    };
    SegNode timeSegment[35105];  // 一学期小时数(20*7*24)*4 true代表未被占用
    void pushUp(int index);
    void pushDown(int index);
    // start-end:要查询的区间 curLeft-curRight:
    // 当前递归到的区间边界 index:当前时间段对应的数组下标
    bool query(int start, int end, int curLeft, int curRight, int index);
    // start-end:要查询的区间 curLeft-curRight:当前递归到的区间边界
    // state:更新后的状态 index:当前时间段对应的数组下标
    void update(int start, int end, bool state, int curLeft, int curRight,
                int index);
    void update(int start, int end, bool state, int curLeft, int curRight,
                int index, itinerary* p);

 public:
    segmentTree();
    segmentTree(const segmentTree& other);
    bool insert(Pair<Time, Time> t, itinerary* p);  // 插入一个新日程，并解决冲突
    void remove(Pair<Time, Time> t);  // 删除一个时间段内的所有日程
    void print(int index);  // 调试用，从左至右输出整棵线段树的日程，没有去重
    void search_time_seg(int index, int ul, int ur, int l,
                         int r, itinerary* ans, int *asize);
    // 查询接口，可以查询一个时间段内的所有日程，返回ans数组，按时间先后有序
};
```

关键函数分析

（1）区间查询：bool query(int start, int end, int curLeft, int curRight, int index);

如果当前查询的区间被包含于当前递归到的区间边界，则返回结果，如果尚未被包含于递归到的区间边界，则继续递归查找区间(pushDown)并更新递归区间边界并进行递归查询。

```
bool query(int start, int end, int curLeft, int curRight, int index) {
        if (start <= curLeft && end >= curRight)
            return timeSegment[index];
        else {
            pushDown(index);
            bool res = true;
```

```
            int mid = curLeft + ((curRight - curLeft) >> 1);
            if (start <= mid)
                res = res && query(start, end, curLeft, mid, index << 1);
            if (end > mid)
                res = res && query(start, end, mid + 1, curRight, index << 1 |
1);
            return res;
        }
    }
```

（2）区间更新：void update(int start, int end, bool state, int curLeft, int curRight, int index);

对于区间修改，朴素的想法是用递归的方式一层层修改（类似于线段树的建立），但这样的时间复杂度比较高。使用懒标记后，对于那些正好是线段树节点的区间，我们不继续递归下去，而是打上一个标记，将来要用到它的子区间的时候，再向下传递。更新时，我们是从最大的区间开始，递归向下处理。如果更新区间被包括于当前区间，则更新区间状态，如果未包含于当前区间，则传递lazy标记并递归更新左右区间。

```
void segmentTree::update(int start, int end, bool state,
            int curLeft, int curRight,
            int index, itinerary* p) {
    // printf("%d %d %d\n", index, curLeft, curRight);
    if (start <= curLeft && end >= curRight) {
        lazy[index] = state ? 1 : -1;
        timeSegment[index].value = state ? 1 : 0;
        timeSegment[index].point = state ? NULL : p;
        timeSegment[index].purity = state ? 0 : 1;
    } else {
        pushDown(index);
        int mid = curLeft + ((curRight - curLeft) >> 1);
        if (start <= mid)
            update(start, end, state, curLeft, mid, index << 1, p);
        if (end > mid)
            update(start, end, state, mid + 1, curRight, index << 1 | 1, p);
        pushUp(index);
    }
}
```

此函数还有一个重载，需要传入一个日程类的指针，在区间更新后，会给线段树每个更新的节点增加一个指向对应日程的指针，算法思路与上面相同。

```
void update(int start, int end, bool state, int curLeft, int curRight,
                int index, itinerary* p) {
        if (start <= curLeft && end >= curRight) {
            lazy[index] = state ? 1 : -1;
            timeSegment[index].value = state ? 1 : 0;
            timeSegment[index].point = state ? NULL : p;
            timeSegment[index].purity = state ? 0 : 1;
        } else {
            pushDown(index);
            int mid = curLeft + ((curRight - curLeft) >> 1);
            if (start <= mid)
                update(start, end, state, curLeft, mid, index << 1, p);
            if (end > mid)
                update(start, end, state, mid + 1, curRight, index << 1 | 1, p);
            pushUp(index);
        }
```

```
    }
```

（3）区间插入：bool insert(Pair<Time, Time> t, itinerary* p)

对于一个新日程，插入时应取出它的起止时间，接着询问该时间区间内是否已经存在其他日程：若存在，则说明发生了日程冲突，立即退出，并返回 False；否则修改该区间状态为占用，并为线段树对应节点添加一个指针。

线段树的 insert 函数是外部直接调用的接口函数之一，传入一个占用时间段 t，和一个对应日程的指针 p。线段树首先对时间段 t 进行一次 query，询问这个时间段内是否被占用，如果被占用，返回 false。

如果没有被占用，线段树则调用一次 update 函数，对这个时间段进行更新，更新状态为占用，添加日程指针为 p。返回 true。

```
bool segmentTree::insert(Pair<Time, Time> t, itinerary* p) {
    int start = t.first.calHours();
    int end = t.second.calHours();
    // printf("%d %d\n", start, end);
    if (!query(start, end, 1, 8760, 1)) {
        // printf("*1\n");
        return false;
    } else {
        // printf("Ready to insert!\n");
        update(start, end, false, 1, 8760, 1, p);
        return true;
    }
}
```

（4）区间删除：void remove(Pair<Time, Time> t)

线段树的 remove 函数是外部直接调用的接口函数之一，它的作用是将一个时间段设置为非占用状态。直接调用 update 函数，设置参数为 0 即可。

```
void segmentTree::remove(Pair<Time, Time> t) {
    int start = t.first.calHours();
    int end = t.second.calHours();
    update(start, end, true, 1, 8760, 1);
}
```

（5）时间段查询：search_time_seg(int index, int ul, int ur, int l, int r, itinierary* ans, int* asize)

线段树的外部接口函数之一，作用是查询一段时间内包含哪些日程，按照时间先后顺序返回到数组 ans 中。其中参数 index, ul, ur 是递归使用的参数，分别表示当前线段树上的节点编号，当前节点维护区间的左端点，当前节点维护区间的右端点；参数 l，r是外部传入的查询时间段；参数 ans 和 asize 是传指针的外部数组，用来返回查询答案，ans 是数组起始位置的指针，asize 是数组元素个数。

调用此函数时，从根节点 1 开始搜索，根节点维护的时间段范围为 1 - 8760. 此函数首先判断当前节点维护的区间，是否在查询范围内。若超出查询范围，直接返回。

接着它会判断当前节点维护的时间区间是否属于一个完整的日程。根据之前 update 函数的定义，如果一个节点维护的区间属于一个完整的日程，那么这个节点会存储一个 point 指针指向该日程；否则，这个节点的 point 指针值为 NULL。那么此函数会判断当前节点的 point 是否为 NULL，若非空，说明这个节点属于一个完整的日程，那么判断这个节点对应的日程是否已经被添加进 ans 数组里，若没有，则添加入 ans 数组中。

如果这个节点 point 指针值为 NULL，说明这个节点不属于一个完整的日程，接着递归询问此节点的左右区间。

需要注意的是，这里的 point 指针起到了类似于传统线段树懒惰标记的作用，当我们访问到 point 非空的节点，说明此节点的所有子树节点都属于同一个日程了，直接保存答案返回即可，不用再去访问它的全部子节点。正是这种优化，保证了它查询具有均摊 log 的复杂度。

不过，就算我们已经用懒惰标记避免了对子树的查询，依然可能出现日程重复的情况。不过，因为我们代码实现中，是严格先序遍历整棵线段树的，而线段树的子树天然按照区间从左至右有序，所以我们取得的日程按照时间是严格不降的。因此，每次我们判断日程是否重复时，只需要和 ans 数组保存的上一个日程比较是否相同即可，不用把 ans 数组整个遍历一遍判重，可以节约很多时间。是一个利用线段树性质的小技巧。这个性质也保证了我们返回 ans 数组是按照时间先后严格有序的。

```
void segmentTree::search_time_seg(int index, int ul, int ur, int l,
                    int r, itinerary* ans, int *asize) {
    if (ur < l || ul > r) return;
    // printf("%d %d %d %d %d\n", index, ul, ur, l, r);
    if (timeSegment[index].point) {
        // printf("!!! %d %d %d %d %d\n", index, ul, ur, l, r);
        if ((*asize) == 0 ||
            !(ans[(*asize) - 1] == *timeSegment[index].point)) {
            // printf("Begin to add\n");
            ans[(*asize)] = *timeSegment[index].point;
            // (*timeSegment[index].point).print();
            // ans[(*asize)].print();
            (*asize) = (*asize) + 1;
            // printf("Ended!\n");
        }
        return;
    } else {
        if (ul == ur) return;
        int mid = ul+ ((ur - ul) >> 1);
        // printf("? %d %d %d %d %d %d", ul, ur, ul, mid, mid + 1, ur);
        if ((index << 1) > 35100) return;
        search_time_seg(index << 1, ul, mid,
                        l, r, ans, asize);
        if ((index << 1 | 1) > 35100) return;
        search_time_seg(index << 1 | 1, mid + 1, ur,
                        l, r, ans, asize);
    }
```

（6）测试输出：printf(int index)

这是测试线段树正确性的调试用函数，可以输入一个节点的序号，接着按照时间先后顺序，依次输出以该节点为根节点的子树内所有日程。这里没有添加去重功能，仅是为了方便观察线段树的结构和检验正确性。

```
void segmentTree::print(int index) {
    // for(int i = 1; i < 20001; i++)
    //     printf("%d %d\n", i, timeSegment[i].purity);
    // printf("index: %d purity: %d\n", index, timeSegment[index].purity);
    // printf("%d ", index);
    if (timeSegment[index].purity) {
        itinerary* p = timeSegment[index].point;
        p->print();
        // printf("Ended!\n");
```

```
    } else {
        if ((index << 1) > 35100) return;
        print(index << 1);
        if ((index << 1 | 1) > 35100) return;
        print(index << 1 | 1);
    }
}
```

时间复杂度分析

　　线段树考虑线段树进行区间操作时的过程，如果当前节点被操作区间完全包含，则不会继续向下递归。考虑对区间 L, R 进行操作，若当前节点 $mid < L$，则会向右子树递归，若 $mid \geq R$，则会向左子树递归。若 $L \leq mid < R$，则区间会分为两部分分别向左右子树递归，即区间 $[L, mid]$ 和 $[mid + 1, R]$。两部分情况相似，我们先考虑右半边。若当前节点 $mid \geq R$，则会向左子树递归。否则，只会向右子树递归，因为此节点的左孩子一定被 $[L, R]$ 完全覆盖。左半边同理。因此，对区间操作时，最多只会递归 $2Deep$ 次，其中 $Deep$ 为线段树的深度。线段树的深度是 $O(\log n)$ 的，因此单次区间操作的复杂度也为 $O(\log n)$.

测试用例及结果

我们单独给线段树模块进行了测试，测试代码如下：

```
segmentTree A;
itinerary x[103];

int main() {
    int Q;
    cin >> Q;
    for (int i = 0, l, r; i < Q; i++) {
        int op;
        Pair<Time, Time> t;
        cin >> op;
        if (op == 1) {
            char str[103];
            cin >> l >> r >> str;
            t.first = l;
            t.second = r;
            String y;
            int m = strlen(str);
            for (int i = 0; i < m; i++)
                y.pushBack(str[i]);
            x[i].setName(y);
            x[i].setTime(t);
            if (A.insert(t, x + i))
                puts("Yes!");
            else
                puts("No!");
        } else if (op == 2) {
            cin >> l >> r;
            t.first = l;
            t.second = r;
            A.remove(t);
        } else if (op == 3) {
            A.print(1);
        }
    }
    system("pause");
```

```
    return 0;
}
```

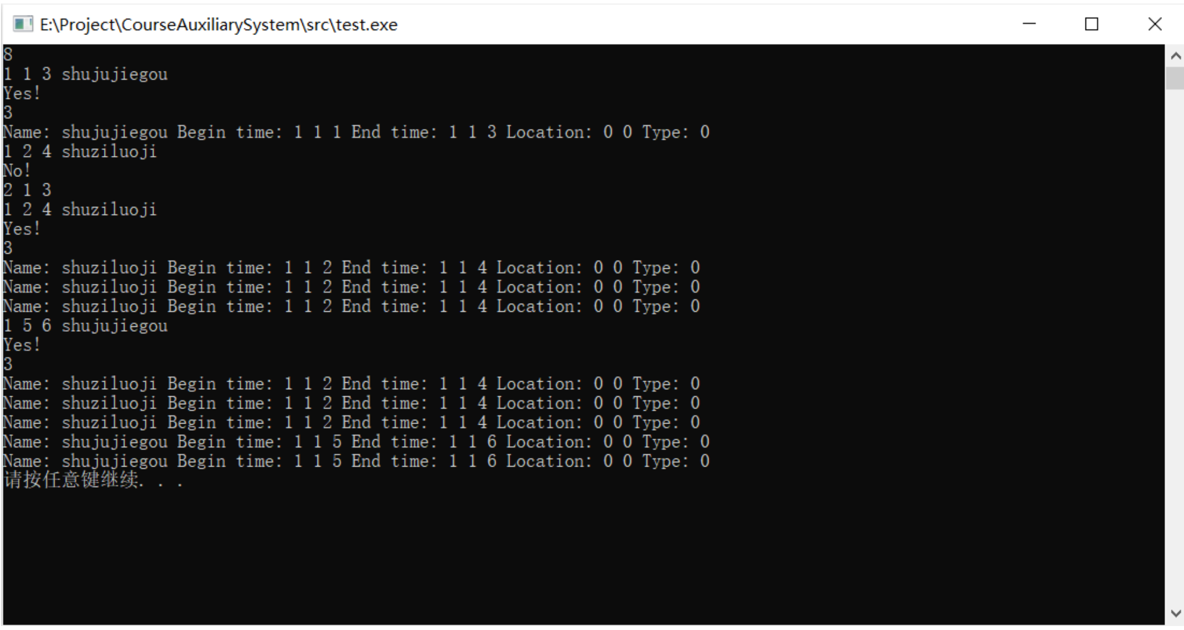
输入:

```
8
1 1 3 shujujiegou
3
1 2 4 shuziluoji
2 1 3
1 2 4 shuziluoji
3
1 5 6 shujujiegou
3
```

输出:

```
Yes!
Name: shujujiegou Begin time: 1 1 1 End time: 1 1 3 Location: 0 0 Type: 0
No!
Yes!
Name: shuziluoji Begin time: 1 1 2 End time: 1 1 4 Location: 0 0 Type: 0
Name: shuziluoji Begin time: 1 1 2 End time: 1 1 4 Location: 0 0 Type: 0
Name: shuziluoji Begin time: 1 1 2 End time: 1 1 4 Location: 0 0 Type: 0
Yes!
Name: shuziluoji Begin time: 1 1 2 End time: 1 1 4 Location: 0 0 Type: 0
Name: shuziluoji Begin time: 1 1 2 End time: 1 1 4 Location: 0 0 Type: 0
Name: shuziluoji Begin time: 1 1 2 End time: 1 1 4 Location: 0 0 Type: 0
Name: shujujiegou Begin time: 1 1 5 End time: 1 1 6 Location: 0 0 Type: 0
Name: shujujiegou Begin time: 1 1 5 End time: 1 1 6 Location: 0 0 Type: 0
```

运行结果:



堆

说明

堆是一个关键字较大（或较小）元素先出的优先队列，它是一棵二叉树，每一个节点都大于（或小于）它的两个孩子节点。堆支持插入、删除、取堆顶元素等操作。其定义如下：

```
template <class T, typename F = Less<T> >
class Heap {
 private:
    Vector<T> data;
    int size;
    void swap(int x, int y);

 public:
    Heap();
    void push(T x);
    T top() const;
    bool isEmpty() const;
    int getSize() const;
    void pop();
};
```

堆是一个模板类，需要传入数据类型 T，和比较方法 F。默认比较方法 F 是小于，其他比较方法定义及实现如下：

```
template<class T>
struct Less {
    bool operator() (const T &lhs, const T &rhs) const;
};

template<class T>
struct Greater {
    bool operator() (const T &lhs, const T &rhs) const;
};

template<class T>
struct LessEqual {
    bool operator() (const T &lhs, const T &rhs) const;
};

template<class T>
struct GreaterEqual {
    bool operator() (const T &lhs, const T &rhs) const;
};
```

堆有两个成员函数。其中，变长数组 data 即为堆中元素所在的数组，是堆实际占用的空间。整形 size 为堆的大小。内部函数 swap 提供堆中两个元素的交换方法。

对外的接口函数包括：将元素插入堆 push，取堆顶元素 top，判空函数 isEmpty，取堆大小 getSize，弹出堆顶元素 pop。下面具体分析比较重要的几个函数：

（1）插入函数 push

插入函数可以将一个元素 x，插入堆中，并调整它在堆中的位置，保证插入完成后，堆依然是一个满足性质的二叉树。其插入过程分为两步：

- 插入元素初始放在堆尾，堆的长度加一；
- 对插入元素进行“上浮”。插入元素与其父亲结点比较大小，若比其父 亲结点大（或小），则与其父亲结点交换。再考查此结点与其新的父亲结点的大小关系， 直至到达堆顶。

代码实现如下：

```
template <class T, typename F>
void Heap<T, F>::push(T x) {
    size++;
    data.pushBack(x);
    int i = size;
    while (i > 1 && F()(data[i >> 1], data[i])) {
        swap(i, i >> 1);
        i = i >> 1;
    }
}
```

（2）取堆顶函数 top

堆顶就是二叉树的根节点，因此只需要取出当前根节点对应的数据即可，代码实现如下：

```
template <class T, typename F>
T Heap<T, F>::top() const {
    return data[1];
}
```

（3）弹出堆顶元素函数 pop

弹出当前堆顶的元素，并调整堆中其余元素的位置，保证弹出之后，堆依然是一个符合条件的二叉树。弹出操作分为两步：

- 用堆尾元素替换堆顶元素，size - 1
- 对此元素进行“下沉”操作。每次比较当前元素和左右孩子的大小关 系，若比某左右孩子的最大值小（或大），则与此最大值对应的左右孩子进行交换。直至 比左右孩子都要大（或小）。

代码实现如下：

```
template <class T, typename F>
void Heap<T, F>::pop() {
    if (size == 0) return;
    data[1] = data[size];
    data.popBack();
    size--;
    int i = 1, j;
    if ((i << 1) > size) return;
    if (((i << 1) | 1) > size || F()(data[(i << 1) | 1], data[i << 1]))
        j = i << 1;
    else
        j = (i << 1) | 1;
    while (j <= size && F()(data[i], data[j])) {
        swap(i, j);
        i = j;
        if ((i << 1) > size) return;
        if (((i << 1) | 1) > size || F()(data[(i << 1) | 1], data[i << 1]))
            j = i << 1;
        else
            j = (i << 1) | 1;
```

```
    }
}
```

其余函数较为简单，在此处略去，具体参见源程序部分。

时空复杂度分析

- 空间复杂度：堆排序只使用了一个额外空间进行交换，因此空间复杂度 O(i)
- 时间复杂度：堆为一棵完全二叉树，因此其树高不超过 O(logn)，每次插入删除 会从叶子走到根或从根走到叶子，因此单次插入删除的时间复杂度为 O(logn)，总 共会进行 n 次插入与删除，因此总时间复杂度为 O(nlogn)

测试用例及结果

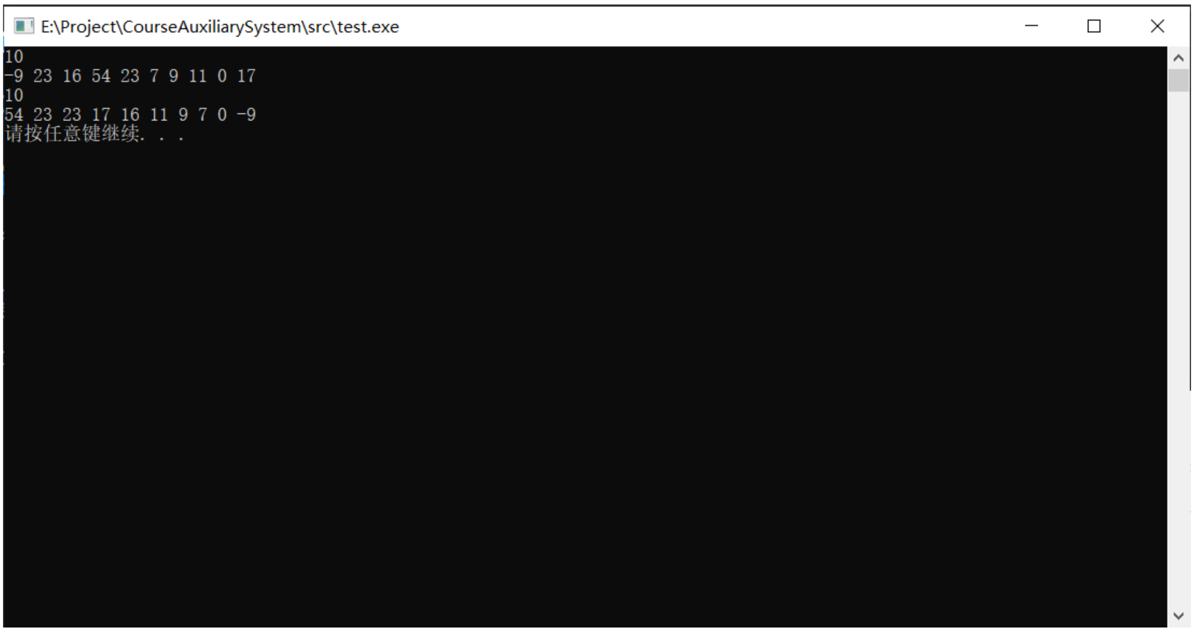
我们以整型数据为例对堆进行了测试，代码如下：

```
Heap<int> heap;

int main() {
    int n;
    cin >> n;

    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        heap.push(x);
    }
    cout << heap.getSize() << endl;
    for (int i = 1; i <= n; i++) {
        cout << heap.top() << " ";
        heap.pop();
    }
    cout << endl;
    return 0;
}
```

输入：

```
10
-9 23 16 54 23 7 9 11 0 17
```

运行结果：



AVL 树

算法描述

AVL树是一种基本的自平衡二叉搜索树，其主要特点为，任何节点的左右子树高度差严格小于等于1。

在本项目中，使用 AVL 树维护所有的课程，比较关键字为课程的名称。由于二叉搜索树的特点，我们可以很方便地在树上按名称搜索课程，又因为它是一棵严格的平衡树，所以每一次插入、查找的时间复杂度都较优，不用担心二叉搜索树在动态插入的过程中出现退化的问题。

值得一提的是，由于我们同时实现了红黑树，所以此处 AVL 树的功能全部可以被红黑树替代，且红黑树拥有随机数据下更好的期望效率。但出于尽可能尝试更多样的数据结构的考虑，我们依然在维护课程时候保留了 AVL 树。

AVL 树的模板类定义：

```
template <class T>
class AVL {
 private:
    int size;
    class AVLNode {
     public:
        T data;
        int size, deep;
        AVLNode* left;
        AVLNode* right;
        AVLNode();
        explicit AVLNode(T dt);
        AVLNode(int s, int d, T dt);
        AVLNode(int s, int d, T dt, AVLNode* l, AVLNode* r);
        AVLNode(const AVLNode &other);
        ~AVLNode();
        int getSize(AVLNode *x);
        int getDeep(AVLNode *x);
        int getSize();
        int getDeep();
        AVLNode* ls();
```

```
        AVLNode* rs();
    };
    AVLNode* root;
    void update(AVLNode *x);
    void leftLeft(AVLNode** u);
    void rightRight(AVLNode** u);
    void leftRight(AVLNode** u);
    void rightLeft(AVLNode** u);

 public:
    AVL();
    ~AVL();
    AVLNode** getRoot();
    void Free(AVLNode** u);
    bool insert(AVLNode** u, T d);
    bool exist(AVLNode* u, T d);
    T search(AVLNode* u, T d);
    bool Delete(AVLNode** u, T d);
    void show_AVL(AVLNode* u);
};
```

在 AVL 类的内部，我们定义了一个子类 AVLNode，其含义是 AVL 树的节点类型。AVLNode 类的成员包括：存储的数据 data，所在子树的大小 size，所在子树的深度 deep，左子树指针 left，右子树指针 right。AVLNode 类的接口函数包括：getSize() 获得子树的大小，getDeep() 获取子树的深度，ls() 获得左子树指针，rs() 获得右子树指针。getSize() 和 getDeep() 根据调用方式可能有所不同进行了重载，实际功能没有太大区别。

在完成定义 AVLNode 类之后，我们可以定义好 AVL 树的所有成员变量：包括一个根节点指针 root 和树大小 size，其他节点通过节点间的指针关系来进行管理。

关键函数与接口介绍：

（1）更新节点：void update(AVLNode* x)

更新指针 x 所指向的节点的 size 和 deep 值。节点 *x 的大小 size 为左右子树 size 相加再加上节点 *x 自身；节点 *x 的深度 deep 为左右子树 deep 的最大值再加 1.

代码实现为：

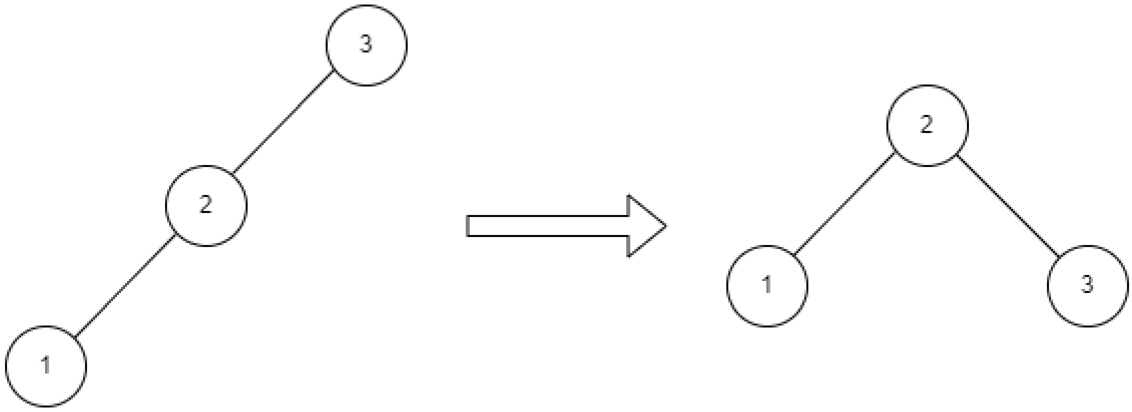
```
template <class T>
void AVL<T>::update(AVLNode *x) {
    if (x == NULL) return;
    x->size = x->getSize(x->left) + x->getSize(x->right) + 1;
    x->deep = Basic :: getMax<int>(x->getDeep(x->left),
                                x->getDeep(x->right)) + 1;
}
```

（2）旋转操作

在插入或删除节点后，一棵原本平衡的二叉搜索树，可能会变得不平衡。为了实现树的自平衡，AVL树通过旋转操作来保证自平衡。

二叉树不平衡的情况有如下四种：

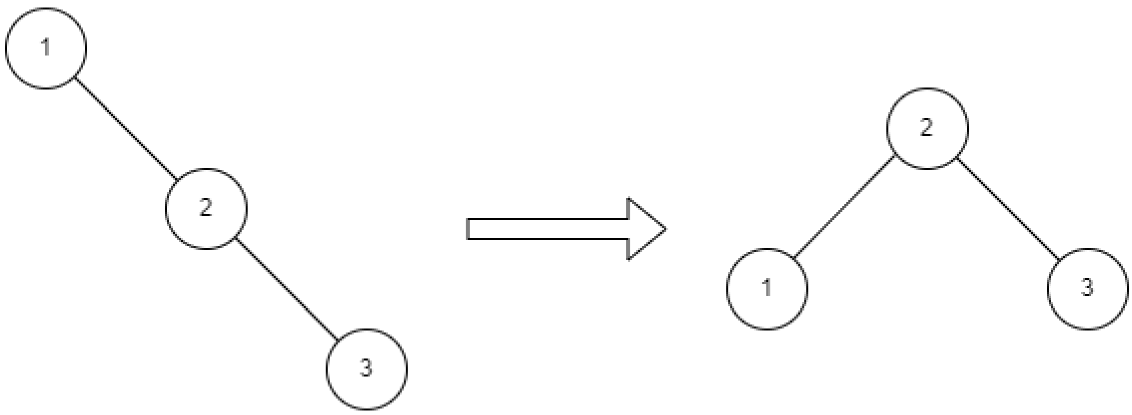
- 左左



右旋一次即可

```
template <class T>
void AVL<T>::leftLeft(AVLNode** u) {
    AVLNode* v = (*u) -> left;
    (*u) -> left = v -> right;
    v -> right = *u;
    update(*u);
    update(v);
    *u = v;
}
```

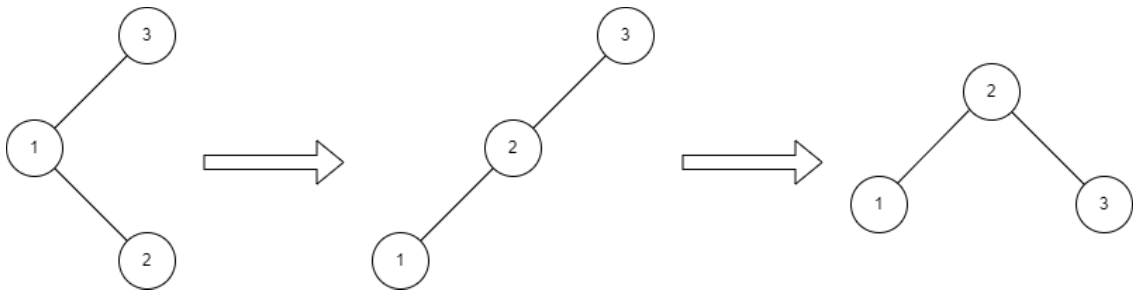
- 右右



左旋一次即可

```
template <class T>
void AVL<T>::rightRight(AVLNode** u) {
    AVLNode* v = (*u) -> right;
    (*u) -> right = v -> left;
    v -> left = *u;
    update(*u);
    update(v);
    *u = v;
}
```

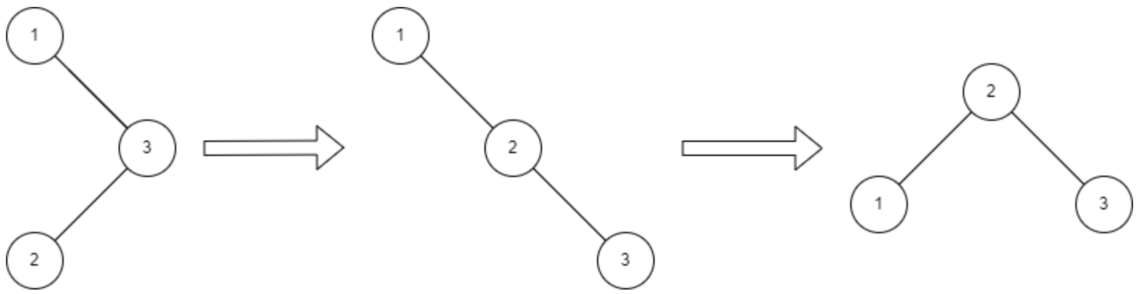
- 左右



左右的情况稍微复杂一些，要先进行一次右旋，将其转化为左左，再进行一次左旋

```
template <class T>
void AVL<T>::leftRight(AVLNode** u) {
    rightRight(&((*u) -> left));
    leftLeft(u);
}
```

- 右左



右左的情况类似左右，要先进行一次左旋，将其化为右右，再进行一次右旋

```
template <class T>
void AVL<T>::rightLeft(AVLNode** u) {
    leftLeft(&((*u) -> right));
    rightRight(u);
}
```

这些旋转操作保证了 AVL 树的自平衡特性。

（3）插入函数：bool AVL :: insert(AVLNode ** u, T d)

在平衡树中插入数据 d，如果成功插入返回 0，如果已经存在了数据 d，那么不重复插入，返回 1。插入时，首先判断当前节点数据和待插入数据是否相等，若相等返回 1。若不相等，比较当前节点数据和待插入数据的大小关系，若大于，则递归在右子树插入此数据；若小于，则递归在左子树插入此数据。直到某节点为空，则将该节点替换为此数据项。

递归返回时，应判断左右子树的平衡性是否保持，若左右子树不平衡，那么根据上述分类的四种情形，进行相应的旋转操作，使子树恢复平衡。

代码实现为：

```
template <class T>
bool AVL<T>::insert(AVLNode** u, T d) {
    if (*u == NULL) {
        *u = new AVLNode(d);
        return 0;
    }
    if (d == (*u)->data) return 1;
```

```
    if (d < (*u)->data) {
        bool f = insert(&((*u)->left), d);
        if (f) return 1;
        update(*u);
        if ((*u)->getDeep((*u)->left) - (*u)->getDeep((*u)->right) == 2) {
            if (d < (*u)->left->data) leftLeft(u);
            else
                leftRight(u);
        }
    } else {
        bool f = insert(&((*u)->right), d);
        if (f) return 1;
        update(*u);
        if ((*u)->getDeep((*u)->right) - (*u)->getDeep((*u)->left) == 2) {
            if (d < (*u)->right->data) rightLeft(u);
            else
                rightRight(u);
        }
    }
    update(*u);
    return 0;
}
```

（4）判断存在函数：bool exist(AVLNode* u, T d)

判断 AVL 树中是否存在值为 d 的数据项。根据 AVL 树二叉搜索树的性质，只需要根据 d 在 AVL 树上搜索即可。首先判断 d 与当前节点的数据是否相等，若相等，则返回 1；若当前节点为空，则表示 AVL 树上不存在以 d 为数据的节点，查找失败，返回 1。

如果 d 比当前节点的数据小，那么递归查找当前节点的左子树；如果 d 比当前节点的数据大，那么递归查找当前节点的右子树。

代码实现为：

```
template <class T>
bool AVL<T>::exist(AVLNode* u, T d) {
    if (u == NULL) {
        return 0;
    }
    if (d == u->data) return 1;
    if (d < u->data) {
        bool f = exist(u->left, d);
        if (f) return 1;
    } else {
        bool f = exist(u->right, d);
        if (f) return 1;
    }
    return 0;
}
```

（5）查找值函数：T search(AVLNode* u, T d)

在 AVL 树中查找与 d 相等的节点。如果找到返回对应节点的数据，否则返回 d 自身。具体递归思路和 exist 函数一致。

代码实现为：

```
template <class T>
T AVL<T>:: search(AVLNode* u, T d) {
    if (u == NULL) {
        return d;
    }
    if (d == u->data) return u->data;
    if (d < u->data) {
        T f = search(u->left, d);
        return f;
    } else {
        T f = search(u->right, d);
        return f;
    }
    return d;
}
```

（6）删除函数：bool Delete(AVLNode** u, T d)

和大多数平衡树一样，AVL 树的删除较为复杂，需要讨论删除后，会出现什么情形的不平衡情况，依次做旋转操作处理，整个函数是采用递归的方式实现的。

如果 d 小于当前节点的数据，那么在当前节点的左子树进行删除，删除完成后，有可能出现左左、左右两种不平衡情况，判断是否不平衡并做出相应的旋转。

如果 d 大于当前节点的数据，那么在当前节点的右子树进行删除，删除完成后，有可能出现右左、右右两种不平衡情况，判断是否不平衡并做出相应的旋转。

如果 d 恰好需要删除当前的节点。首先判断当前节点的左右子树是否为空，若有子树为空，那么直接用另一个非空子树替代当前节点即可。否则在当前节点的右子树里面找到一个最小的元素，将其赋值给当前节点，再递归地在右子树中删除刚刚找到的最小元素即可。同样，右子树删除后，有可能出现右左、右右两种不平衡情况，判断并进行相应的旋转即可。

代码实现为：

```
template <class T>
bool AVL<T>::Delete(AVLNode** u, T d) {
    if (*u == NULL) return 0;
    if (d < (*u)->data) {
        bool f = Delete(&((*u)->left), d);
        if (f == 0) return 0;
        update(*u);
        if ((*u)->getDeep((*u)->right) - (*u)->getDeep((*u)->left) == 2) {
            if ((*u)->getDeep((*u)->right->left) > \
                (*u)->getDeep((*u)->right->right))
                rightLeft(u);
            else
                rightRight(u);
        }
    } else {
        if (d > (*u)->data) {
            bool f = Delete(&((*u)->right), d);
            if (f == 0) return 0;
            update(*u);
            if ((*u)->getDeep((*u)->left) -
                (*u)->getDeep((*u)->right) == 2) {
                if ((*u)->getDeep((*u)->left->right) > \
                    (*u)->getDeep((*u)->left->left))
                    leftRight(u);
```

```
                else
                    leftLeft(u);
            }
        } else {
            if ((*u)->left == NULL || (*u)->right == NULL) {
                if ((*u)->left) {
                    (*u) = (*u)->left;
                    return 1;
                }
                if ((*u)->right) {
                    *u = (*u)->right;
                    return 1;
                }
                *u = NULL;
                update(*u);
                return 1;
            } else {
                AVLNode* v = (*u)->right;
                while (v->left) v = v->left;
                (*u)->data = v->data;
                Delete(&((*u)->right), v->data);
                update(*u);
                if ((*u)->getDeep((*u)->left) -
                    (*u)->getDeep((*u)->right) == 2) {
                    if ((*u)->getDeep((*u)->left->right) > \
                        (*u)->getDeep((*u)->left->left))
                        leftRight(u);
                    else
                        leftLeft(u);
                }
            }
        }
    }
    update(*u);
    return 1;
}
```

因为根据代码规范要求不允许直接传引用，所以指针操作写得有点复杂。

时间复杂度分析

当根据 BST 规则进行操作时，AVL的平衡性可能被破坏，显然，这种平衡性的破坏仅涉及到该节点的祖先。对于插入操作，当插入一个节点后，从祖父开始，该节点的每个祖先都有可能失衡，且可能不止一个；对于删除操作，当删除一个节点后，从父亲开始，每个祖先都可能失衡，但只可能有一个失衡。为了解决失衡问题，必须进行旋转操作。

由前述分析可知，AVL 的所有旋转操作都在局部进行，只涉及几个节点的指针交换。因此，每一次旋转复杂度都为 O(1)，由之前分析可知，在每一深度只需要检测并旋转至多 1 次。故总复杂度为 O(log n)

值得一提的是，虽然删除操作时，AVL 只有至多一个节点失衡，但是其依然需要 O(log n) 的复杂度，这是由于 AVL 在调整失衡节点时，可能造成更高祖先的失衡，失衡会从第一个不平衡节点开始向上传递，最多造成 log n 个节点先后失衡。故删除复杂度 O(log n)

模块测试用例及结果

对于 AVL 树模块，我们以整型数据为例进行了测试，测试代码如下：

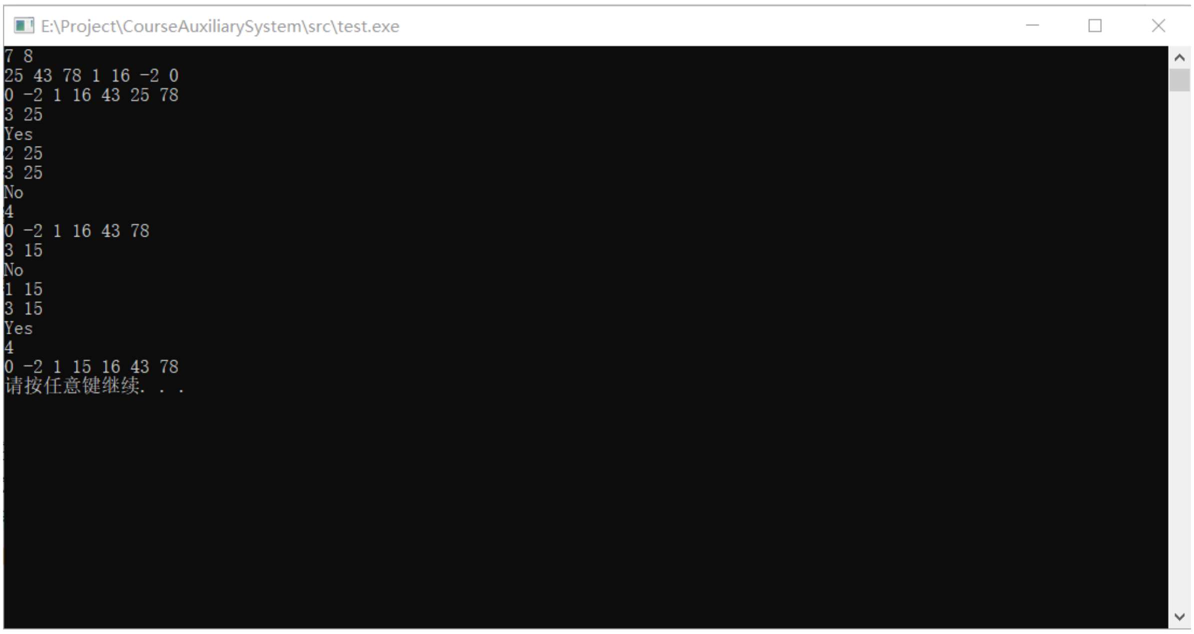
```
#include "AVL.cpp"
#include <bits/stdc++.h>
using namespace std;

AVL<int> avl;

int main() {
    int n, q;
    cin >> n >> q;
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        avl.insert(avl.getRoot(), x);
    }
    avl.show_AVL(*avl.getRoot());
    cout << endl;
    avl.show_AVL_2(*avl.getRoot());
    cout << endl;
    for (int i = 0; i < q; i++) {
        int x;
        cin >> x;
        if (x == 1) {
            int y;
            cin >> y;
            avl.insert(avl.getRoot(), y);
        } else if (x == 2) {
            int y;
            cin >> y;
            avl.Delete(avl.getRoot(), y);
        } else if (x == 3) {
            int y;
            cin >> y;
            cout << (avl.exist(*avl.getRoot(), y) ? "Yes" : "No") << endl;
        } else if (x == 4) {
            avl.show_AVL(*avl.getRoot());
            cout << endl;
            avl.show_AVL_2(*avl.getRoot());
            cout << endl;
        }
    }
    system("pause");
}
```

测试用的数据如下：

```
7 8
25 43 78 1 16 -2 0
3 25
2 25
3 25
4
3 15
1 15
3 15
4
```

测试结果如下：

运行结果如下图。测试时，首先插入了 7 个乱序的整数，插入到AVL树中后，输出AVL树中序遍历结果，输出的结果有序。随后进行 8 次操作，第一次询问 25 是否在AVL树中，由于之前已经插入，得到结果 Yes。第二次操作删去 25，第三次操作询问 25 是否在AVL树中，由于已删除，得到结果 No。第四次操作输出 AVL 树的中序遍历结果，共 6 个数据，结果有序。第五次操作询问 15 是否在AVL树中，此时15 尚未插入，得到结果 No。第六次操作将 15 插入到AVL树中，第七次操作询问 15 是否在红黑树中，得到结果 Yes。最后输出AVL树中序遍历结果，共 7 个数据，结果有序。



红黑树

在求出每个文件的信息后，我们以 MD5 码为关键字，用红黑树进行维护，来实现文件的去重。红黑树是一种效率较高的自平衡二叉查找树，广泛用于Linux 的进程管理、内存管理，设备驱动及虚拟内存跟踪等一系列工程实践中。核心思想是在二叉查找树中引入了节点颜色的概念，并规定了一系列性质，在插入删除时维护这些性质，达到高效查找、维护的目的，统计性能优于 AVL 树等数据结构。

红黑树的定义

红黑树是一棵二叉搜索树，满足二叉搜索树的所有性质，在此基础上，红黑树每个节点都有一个颜色属性，并满足如下性质：

- 节点的颜色为红色或黑色
- 根节点是黑色
- 不能有两个连续的红色节点（即红色节点的孩子必须是黑色）
- 从任一节点到其子树内每个叶子的路径上的黑色节点数量相同

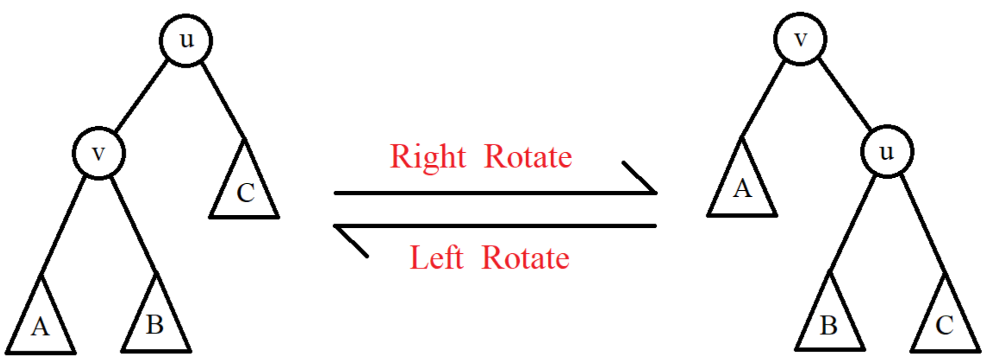
类声明如下:

```
enum RBTColor {RED, BLACK};

template<class T>
struct RBTNode {
    T data;
    RBTColor color;
    RBTNode *parent;
    RBTNode *left, *right;
    RBTNode();
    RBTNode(T value, RBTColor c, RBTNode *p, RBTNode *l, RBTNode *r);
};

template <class T>
struct RBTree {
 private:
    RBTNode<T> *root;
    void leftRotate(RBTNode<T> *x);
    void rightRotate(RBTNode<T> *x);
    void insertFixUp(RBTNode<T> *x);

 public:
    RBTree();
    ~RBTree();

    void insert(T data);
    bool find(T key);
};
```

在这里，我们定义了枚举类 RBTColor 表示节点颜色类，该类有两个取值，RED 和 BLACK。这样做可以有效防止非法情况出现。RBTNode 为红黑树的节点类，这里采用模板类方便复用。其属性除了节点数据 data 和颜色 color 外，还有三个指针 parent,left,right，分别指向当前节点在红黑树上的父亲以及左右孩子。

红黑树的旋转

红黑树的旋转与其他带旋自平衡二叉搜索树类似，分为左旋与右旋，如图所示:



平衡树的旋转操作有一个重要的性质就是，旋转前后树的形态发生了变化，但是仍然满足原来二叉搜索树的性质。以右旋为例，上方左图对节点 u 进行右旋操作后，u 的左孩子 v 变为了 u 的父亲，而 v 的右子树挂到了 u 的左子树的位置上，树的形态发生了改变，但仍然满足二叉搜索树的性质。红黑树左旋右旋的代码如下：

```
template<class T>
void RBTree<T>::leftRotate(RBTNode<T> *x) {
    RBTNode<T> *y = x -> right;
    x -> right = y -> left;
    if (y -> left)
        y -> left -> parent = x;
    y -> left = x;
    y -> parent = x -> parent;
    if (x -> parent) {
        if (x -> parent -> left == x)
            x -> parent -> left = y;
        else
            x -> parent -> right = y;
        x -> parent = y;
    } else {
        root = y;
        x -> parent = y;
    }
}

template<class T>
void RBTree<T>::rightRotate(RBTNode<T> *x) {
    RBTNode<T> *y = x -> left;
    x -> left = y -> right;
    if (y -> right)
        y -> right -> parent = x;
    y -> right = x;
    y -> parent = x -> parent;
    if (x -> parent) {
        if (x -> parent -> left == x)
            x -> parent -> left = y;
        else
            x -> parent -> right = y;
        x -> parent = y;
    } else {
        root = y;
        x -> parent = y;
    }
}
```

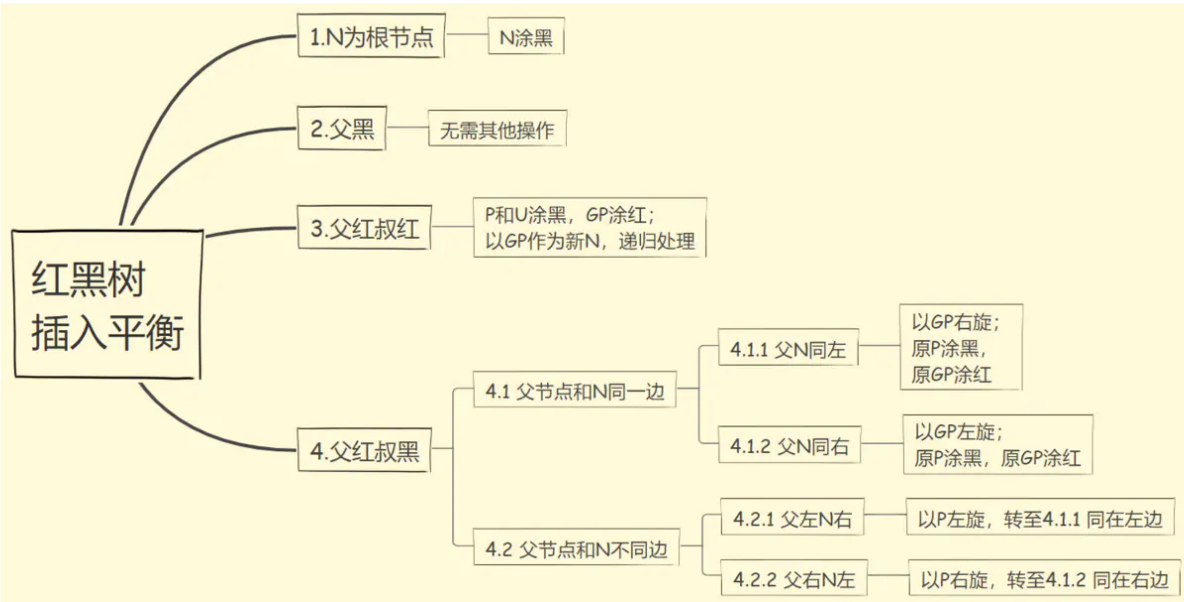
红黑树的插入

将一个节点插入红黑树中，首先需要在二叉搜索树上找到需要插入的位置，将节点插入。随后，将节点颜色设为红色，来保证性质 4，随后，我们需要通过一系列的调整使其重新成为一棵红黑树。代码如下：

```
template<class T>
void RBTree<T>::insert(T data) {
    RBTNode<T> *node = NULL;
    if (!(node = new RBTNode<T>(data, RBTColor::BLACK, NULL, NULL, NULL)))
        return;
```

```
    RBTNode<T> *v = NULL;
    RBTNode<T> *u = root;
    while (u) {
        v = u;
        if (node -> data < u -> data)
            u = u -> left;
        else
            u = u -> right;
    }
    if (v) {
        node -> parent = v;
        if (node -> data < v -> data)
            v -> left = node;
        else
            v ->  right = node;
    } else {
        node -> parent = v;
        root = node;
    }
    node -> color = RBTColor::RED;
    insertFixUp(node);
}
```

考虑直接插入后，红黑树的哪些性质会被打破。首先，性质1仍然满足，性质4由于新插入节点为红色，也依然满足。会被打破的只有性质2与性质3。若性质2被打破，则当前节点为根节点，直接改为黑色即可。若性质3被打破，则当前节点的父亲也是红色。接下来如下图分情况处理，直至所有性质都被满足。



代码如下：

```
template<class T>
void RBTree<T>::insertFixUp(RBTNode<T> *node) {
    RBTNode<T> *parent;
    RBTNode<T> *grandParent;
    while ((parent = node -> parent) && parent -> color == RBTColor::RED) {
        grandParent = parent -> parent;
        if (parent == grandParent -> left) {
            RBTNode<T> *uncle = grandParent -> right;
            if (uncle && uncle -> color == RBTColor::RED) {
                uncle -> color = RBTColor::BLACK;
```

```
                parent -> color = RBTColor::BLACK;
                grandParent -> color = RBTColor::RED;
                node = grandParent;
                continue;
            }

            if (parent -> right == node) {
                RBTNode<T> *tmp;
                leftRotate(parent);
                tmp = parent;
                parent = node;
                node = tmp;
            }

            parent -> color = RBTColor::BLACK;
            grandParent -> color = RBTColor::RED;
            rightRotate(grandParent);
        } else {
            RBTNode<T> *uncle = grandParent -> left;
            if (uncle && uncle -> color == RBTColor::RED) {
                uncle -> color = RBTColor::BLACK;
                parent -> color = RBTColor::BLACK;
                grandParent -> color = RBTColor::RED;
                node = grandParent;
                continue;
            }

            if (parent -> left == node) {
                RBTNode<T> *tmp;
                rightRotate(parent);
                tmp = parent;
                parent = node;
                node = tmp;
            }

            parent -> color = RBTColor::BLACK;
            grandParent -> color = RBTColor::RED;
            leftRotate(grandParent);
        }
    }
    root -> color = RBTColor::BLACK;
}
```

红黑树的查找

为了处理文件去重，我们还需要在红黑树上查询某个文件，此时需要利用二叉搜索树的性质，这一部分在 AVL 树一节中已有详细描述，在此不多赘述，代码如下：

```cpp
template<class T>
bool RBTree<T>::find(T key) {
    RBTNode<T> *u = root;
    while (u) {
        if (key == u -> data)
            return true;
        if (key < u -> data)
            u = u -> left;
        else
            u = u -> right;
    }
    return false;
}
```

时间复杂度分析

红黑树每次插入并调整后，都满足我们之前所说的性质。不妨设深度最浅的叶子（空节点）深度为 x，深度最深的叶子深度为 y，由性质 4，根到两个叶子的路径上的黑色节点数量相同，又由性质三，红色点不能连续出现，因此 $y <= 2x$。设红黑树总节点数为 n，则有 $2^x - 1 < n \le 2^y - 1$，可得 $y \le 2\log n$。因此，红黑树高度不超过 2logn，单次插入、查询的时间复杂度均为 O(logn)

模块测试用例及结果

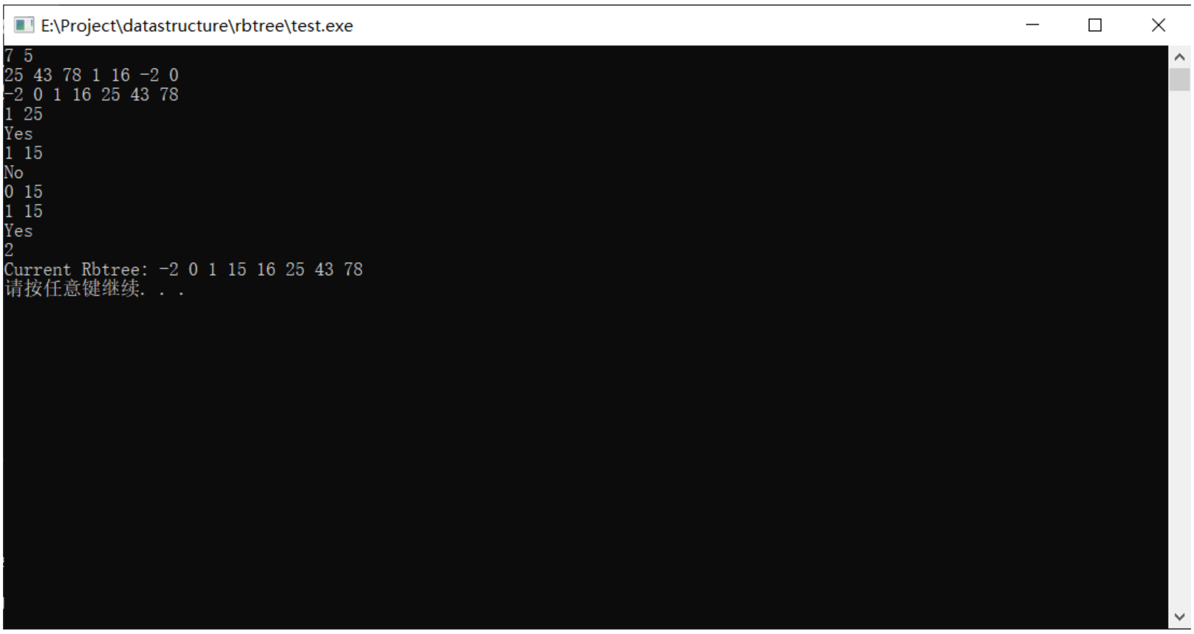
对于红黑树模块，我们以整型数据为例进行了测试，测试代码如下：

```cpp
#include "rbtree.hpp"
using namespace std;

RBTree<int> A;

int main() {
    int n, q;
    cin >> n >> q;
    for (int i = 1, x; i <= n; i++) {
        cin >> x;
        A.insert(x);
    }
    A.print();
    for (int op, x; q--; ) {
        cin >> op;
        if (op == 0) {
            cin >> x;
            A.insert(x);
        } else if(op == 1) {
            cin >> x;
            cout << (A.find(x) ? "Yes" : "No") << endl;
        } else {
            cout << "Current Rbtree: ";
            A.print();
            cout << endl;
        }
    }
}
```

测试用的数据如下：

```
7 5
25 43 78 1 16 -2 0
1 25
1 15
0 15
1 15
2
```

运行结果如下图。测试时，首先插入了 7 个乱序的整数，插入到红黑树中后，输出红黑树中序遍历结果，输出的结果有序。随后进行 5 次操作，第一次询问 25 是否在红黑树中，由于之前已经插入，得到结果 Yes。第二次询问 15 是否在红黑树中，此时15 尚未插入，得到结果 No。第三次操作将 15 插入到红黑树中，第四次操作询问 15 是否在红黑树中，得到结果 Yes。最后输出红黑树中序遍历结果，为 -2 0 1 15 16 25 43 78，共 8 个数据，结果有序。



哈夫曼树 Huffmantree

如果要实现文件的压缩，就需要通过哈夫曼树进行压缩和解压。

首先需要定义树节点以及节点的比较方式

```cpp
class TreeNode {
 public:
    unsigned char c;
    int weight;
    TreeNode* left;
    TreeNode* right;
    TreeNode();
    explicit TreeNode(int);
    TreeNode(unsigned char, int);
    ~TreeNode();
};

struct NodeCompare {
    bool operator()(TreeNode* lhs, TreeNode* rhs) const {
        return lhs->weight > rhs->weight;
```

```
    }
};
```

可以看到树节点的比较大小方式是通过比较节点权重大小进行。树节点中记录了节点代表的字符c以及该字符的权重，以及左右结点的指针。

成员函数有构造参数，可以通过传入权重以及字符的方式初始化树节点。

其次完成哈夫曼树的定义：

```
class HuffmanTree {
 private:
    int64_t totalBits;
    int frequency[256];
    TreeNode* root;
    Heap<TreeNode*, NodeCompare> priQueue;
    std::string charCode[256];
    void inorder(TreeNode*, std::string);
    Vector<unsigned char> encode(const Vector<unsigned char>&);
    Vector<unsigned char> decode(const Vector<unsigned char>&);
    void bulidTree(int[], int);
    int64_t getTotalBits() const;
    void clear();

 public:
    HuffmanTree();
    ~HuffmanTree();
    bool upload(const std::string, const std::string);
    bool download(const std::string, const std::string, const std::string);
};
```

在哈夫曼树中，totalbits表示压缩后字符数，frequency是每个字符出现的频率数组，priqueue是通过对实现的优先队列，在建树过程中使用，charCode用于存储字符编码后的01字符串。

成员函数包括：inorder中序遍历哈夫曼树以获取每个字符的编码方案，encode，传入待编码的字符串进行编码，decode，传入待解码的字符串进行解码，bulidTree，传入字符频率数组以及字符数量（默认为256）建立哈夫曼树,clear则是清空哈夫曼树内容。

接口函数为upload上传文件，传入原地址以及目的地址，是对于encode的封装，download下载文件，传入原地址，目的地址，文件名，是对于decode的封装。

算法描述

接下来重点讲解核心函数bulidTree,encode,decode。

buildTree中，先为权重不为0的字符创建树节点并放入堆中，随后当堆的大小大于1时，连续两次取出堆顶元素并创建这两个节点的根节点，根节点权重为两个节点之和，并将新节点压入堆中。

```
void HuffmanTree::bulidTree(int charWeight[], int n = 256) {
    for (int i = 0; i < n; i++) {
        if (charWeight[i] > 0) {
            TreeNode* node = new TreeNode((unsigned char)i, charWeight[i]);
            priQueue.push(node);
        }
    }
    while (priQueue.getSize() > 1) {
        TreeNode* minNode1 = priQueue.top();
```

```
        priQueue.pop();
        TreeNode* minNode2 = priQueue.top();
        priQueue.pop();
        TreeNode* mergeNode = new TreeNode(minNode1->weight + minNode2->weight);
        mergeNode->left = minNode1;
        mergeNode->right = minNode2;
        priQueue.push(mergeNode);
    }
    root = priQueue.top();
    priQueue.pop();
}
```

encode中，我们先遍历待编码字符串获取每个字符出现的频率，随后建立哈夫曼树，并通过中序遍历获取字符编码方案，随后则是遍历待编码字符串一一进行编码，同时记录字符串的总比特数（当编码至末尾时，如果总比特数不为8的整倍数，则需要补上0，因此需要记录总比特数用于解码时判定真正的结尾在哪里）。

```
Vector<unsigned char> HuffmanTree::encode(const Vector<unsigned char>& text) {
    Vector<unsigned char> result;
    int bitCount = 0;
    unsigned char temp = 0;
    std::string emptyString = "";
    for (int i = 0; i < text.getSize(); i++) {
        frequency[text[i]]++;
    }
    bulidTree(frequency);
    inorder(root, emptyString);
    for (int i = 0; i < text.getSize(); i++) {
        std::string temps = charCode[text[i]];
        int length = charCode[text[i]].size();
        for (int j = 0; j < length; j++) {
            if (temps[j] == '0')
                temp = temp << 1;
            else
                temp = temp << 1 | 1;
            ++bitCount;
            ++totalBits;
            if (bitCount == 8) {
                result.pushBack(temp);
                temp = 0;
                bitCount = 0;
            }
        }
    }
    if (bitCount > 0) temp = temp << (8 - bitCount);
    result.pushBack(temp);
    return result;
}
```

在decode中，则是根据已有的哈夫曼树依次遍历比特串，当遇到树叶节点（左右节点为空）时证明完成一个字符的解码，将该字符放入结果数组中，直到totalbits归0证明解压完毕。

```
Vector<unsigned char> HuffmanTree::decode(const Vector<unsigned char>& code) {
    Vector<unsigned char> text;
    int count = 7;
    int index = 0;
```

```
    TreeNode* cur = root;
    // printf("%d\n",code.getSize());
    while (index < code.getSize() && totalBits > 0) {
        int bit = (code[index] & (1 << count)) == 0 ? 0 : 1;
        if (count == 0) {
            ++index;
            count = 8;
        }
        if (bit)
            cur = cur->right;
        else
            cur = cur->left;
        if (cur->right == cur->left) {
            text.pushBack(cur->c);
            cur = root;
        }
        --count;
        --totalBits;
    }
    return text;
}
```

时空复杂度分析

1.编码阶段：

建树主要分为三大部分：（1）建立字符出现次数映射表（2）通过哈希表建立优先队列（3）通过优先队列建哈夫曼树

首先，建立映射表时需要遍历输入字符串，设输入字符串的长度为N，因此这部分的时间复杂度为$O(N)$

其次，设哈希表的元素数量为n，也就是字符串中包含的不重复字符数为n，在建立了优先队列时，堆的单次插入时间复杂度为$O(logn)$，因此总的时间复杂度为$O(nlogn)$

最后，优先队列的元素数量与字符串中包含不重复的字符数相同，也为n，在循环中，我们每次取出两个元素并放入一个元素，也即是说每经历一次循环优先队列的大小都会减小1，直到队列中只剩一个元素，因此我们一共进行了n-1次循环，时间复杂度为$O(n)$

综上所述，建立一个哈夫曼树的总时间复杂度为$O(N+nlogn)$，其中N为输入字符串的长度，n为字符串中不重复的字符数量，而空间复杂度为$O(n)$，因为建立了n个树节点同时还有映射表数组使用的空间

随后则是逐字符编码，该过程的时间复杂度为$O(m)$，m为编码字符串长度

2.解码阶段：

由于我们需要遍历整个字符串，串中每个字符（0或1）都进行了一次递归，而递归函数中也没有循环，因此这一阶段的时间复杂度为$O(N)$，N为输入的待解码字符串的长度，而空间复杂度则是由于我们的递归函数有着系统栈开销，因此总的空间复杂度也为$O(N)$